Testing in Resource Constrained Execution Environments

Gregory M. Kapfhammer
Department of Computer Science
Allegheny College

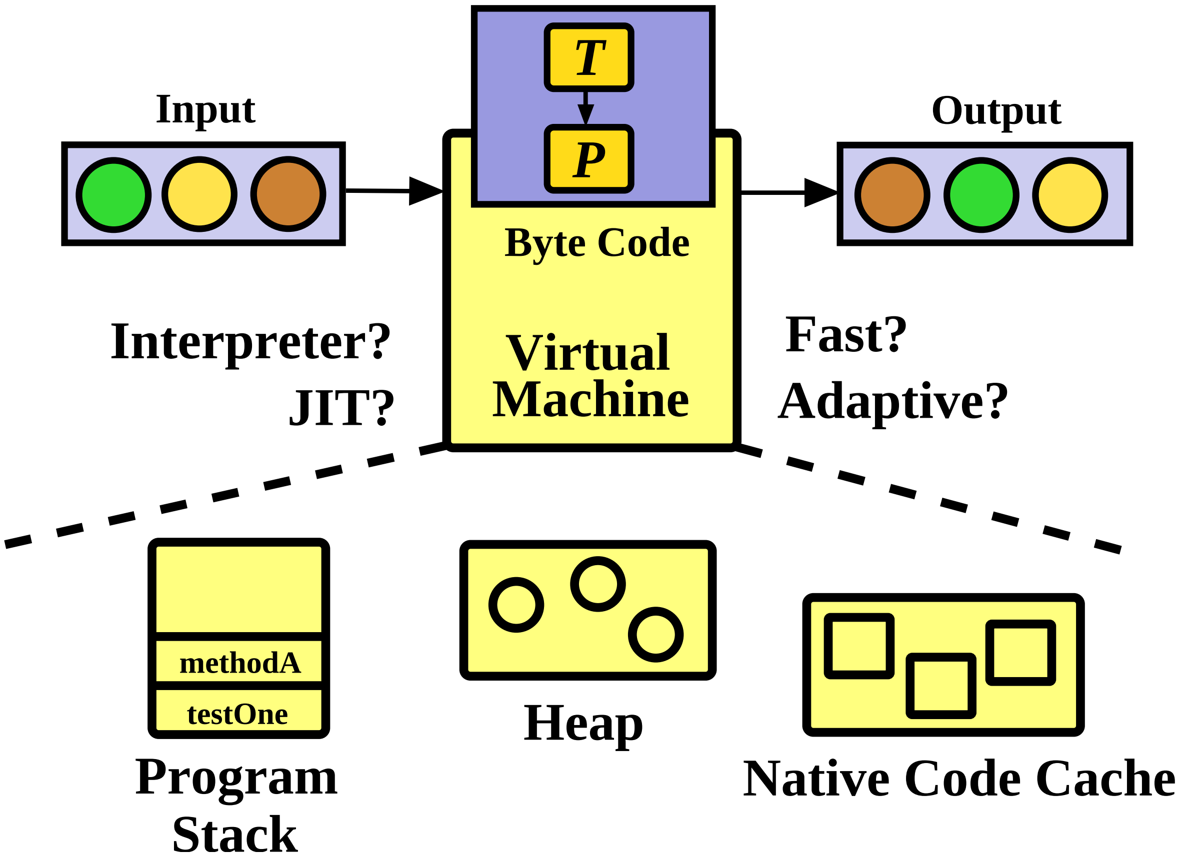
Mary Lou Soffa
Department of Computer Science
University of Virginia

Daniel Mossé
Department of Computer Science
University of Pittsburgh

Contributions

- Use of native code unloading during test suite execution in a resource constrained environment
- Identification of the testing techniques that yield the greatest reduction in execution time and native code size
- Characterization of how software applications and test suites restrict and/or support resource constrained testing
- Cost-benefit analysis for the use of sample-based and exhaustive profiles of program testing behavior
- **Executes test suites faster when memory resources are limited!**

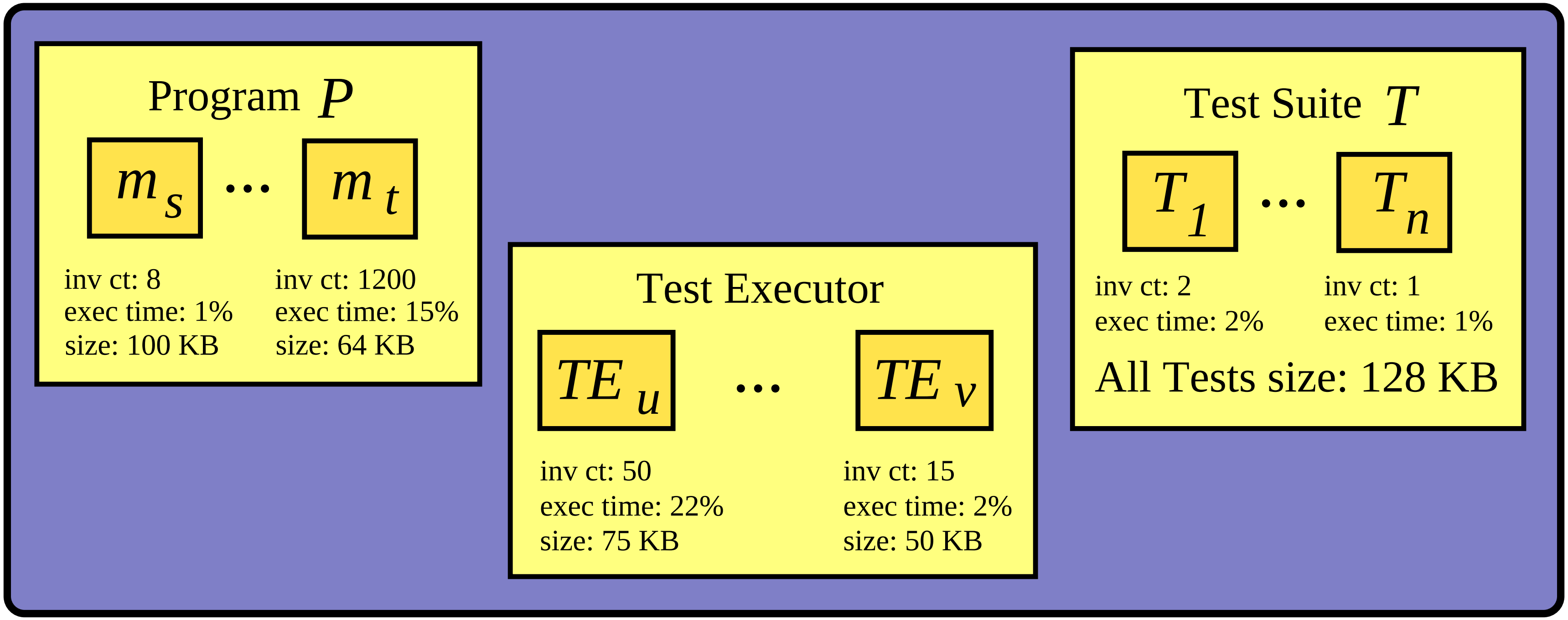
Test Suite Execution with a JVM



- During testing the JVM must manage limited resources

Resource Constrained Testing

Memory Resident Native Code Bodies



- JIT compiler produces native code that exhausts limited memory resources
- Frequent invocation of GC increases testing time

Test Suite Execution Strategies

- **Omit tests?** - Could reduce overall confidence in the correctness of P
- **Use non-constrained environment?** - Defects related to P's interaction with environment might not be isolated
- **Execute tests individually?** - Might increase overall testing time and violate test order dependencies
- **Unload with offline profile?** - Not useful if P and T change frequently during regression testing
- **Our Approach**: Use online behavior profiles to guide the unloading of native code

Experiment Goals and Design

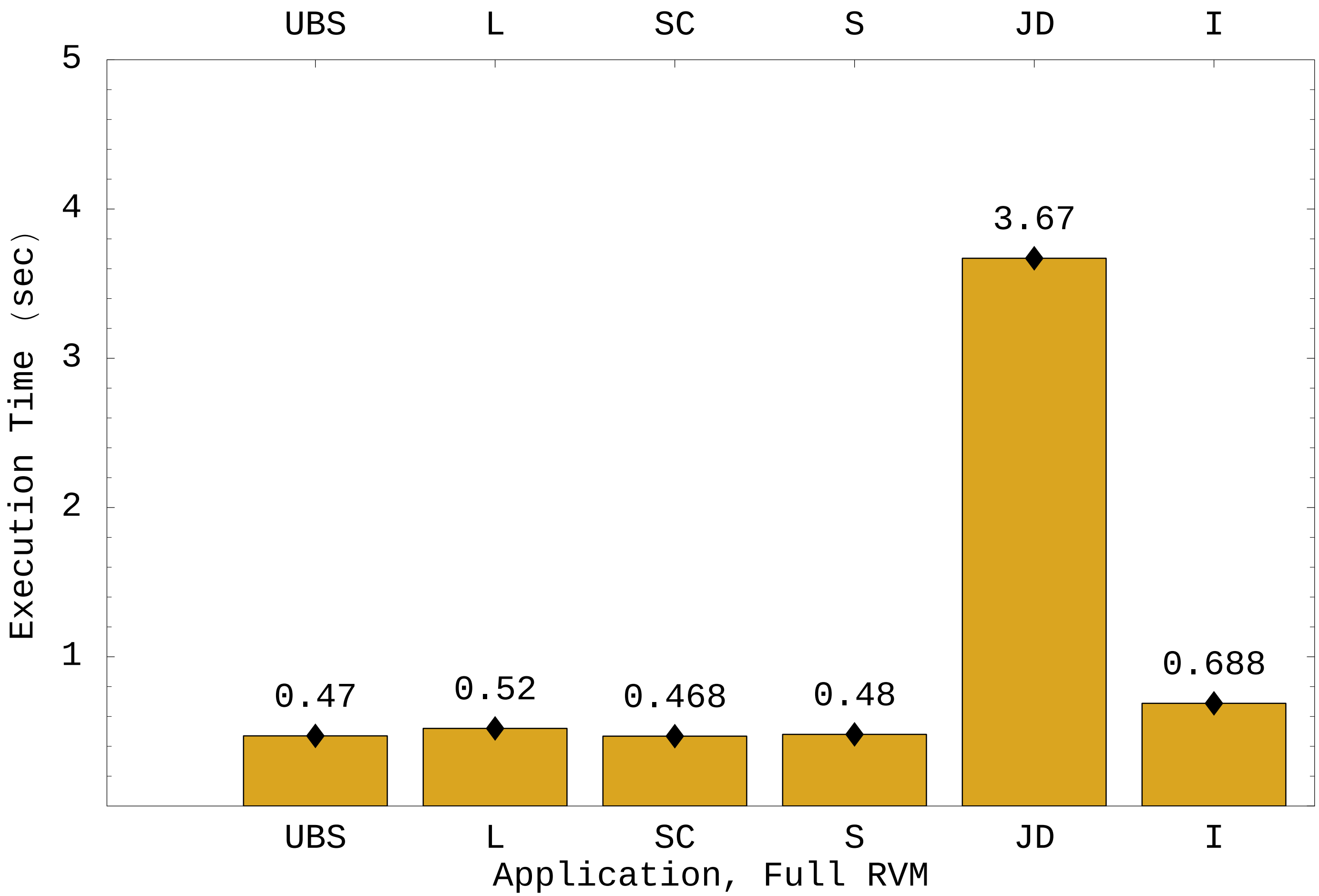
- **Research Question:** Can an adaptive code unloading JVM reduce time and space overheads associated with memory constrained testing?
- **Experiment Metrics**: percent reduction in time, $\mathcal{T}_R^{\%}(P, T)$ and space, $\mathcal{S}_R^{\%}(P, T)$
- Jikes RVM 2.2.1, JUnit 3.8.1, GNU/Linux 2.4.18
- Sample-based (*S*) and exhaustive (*X*) program profi les
- Timer (*TM*), garbage collection (*GC*), and code cache size (*CS*) triggers the unloading technique

Case Study Applications

Name	*Min* Size (MB)	# Tests	NCSS
`UniqueBoundedStack` (UBS)	8	24	362
`Library` (L)	8	53	551
`ShoppingCart` (SC)	8	20	229
`Stack` (S)	8	58	624
`JDepend` (JD)	10	53	2124
`IDTable` (ID)	11	24	315

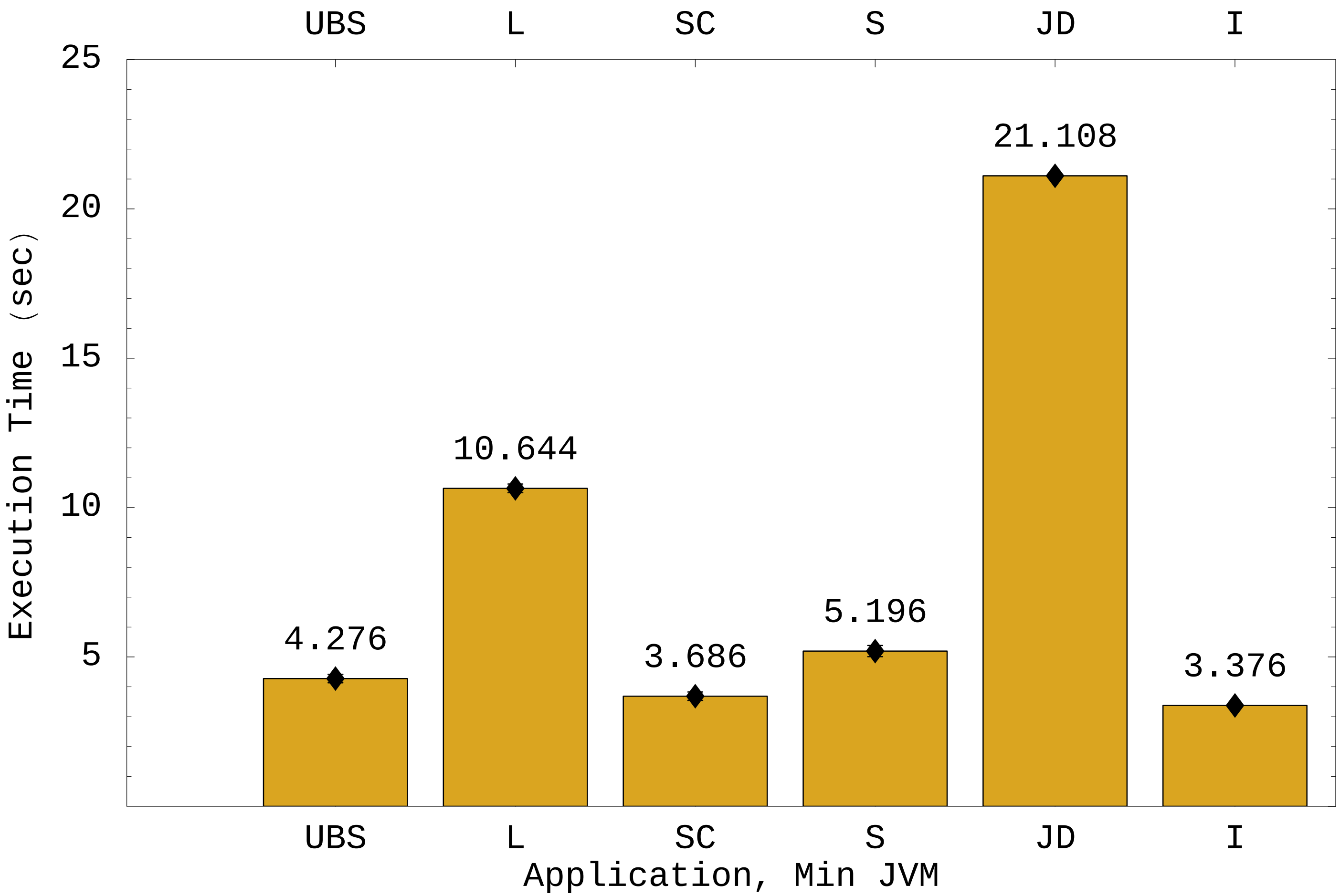
- Empirically determined the *MIN* Jikes RVM heap size

Testing Time Overhead: *Full* RVM



- When memory is not constrained, testing time is acceptable

Testing Time Overhead: *Min* RVM



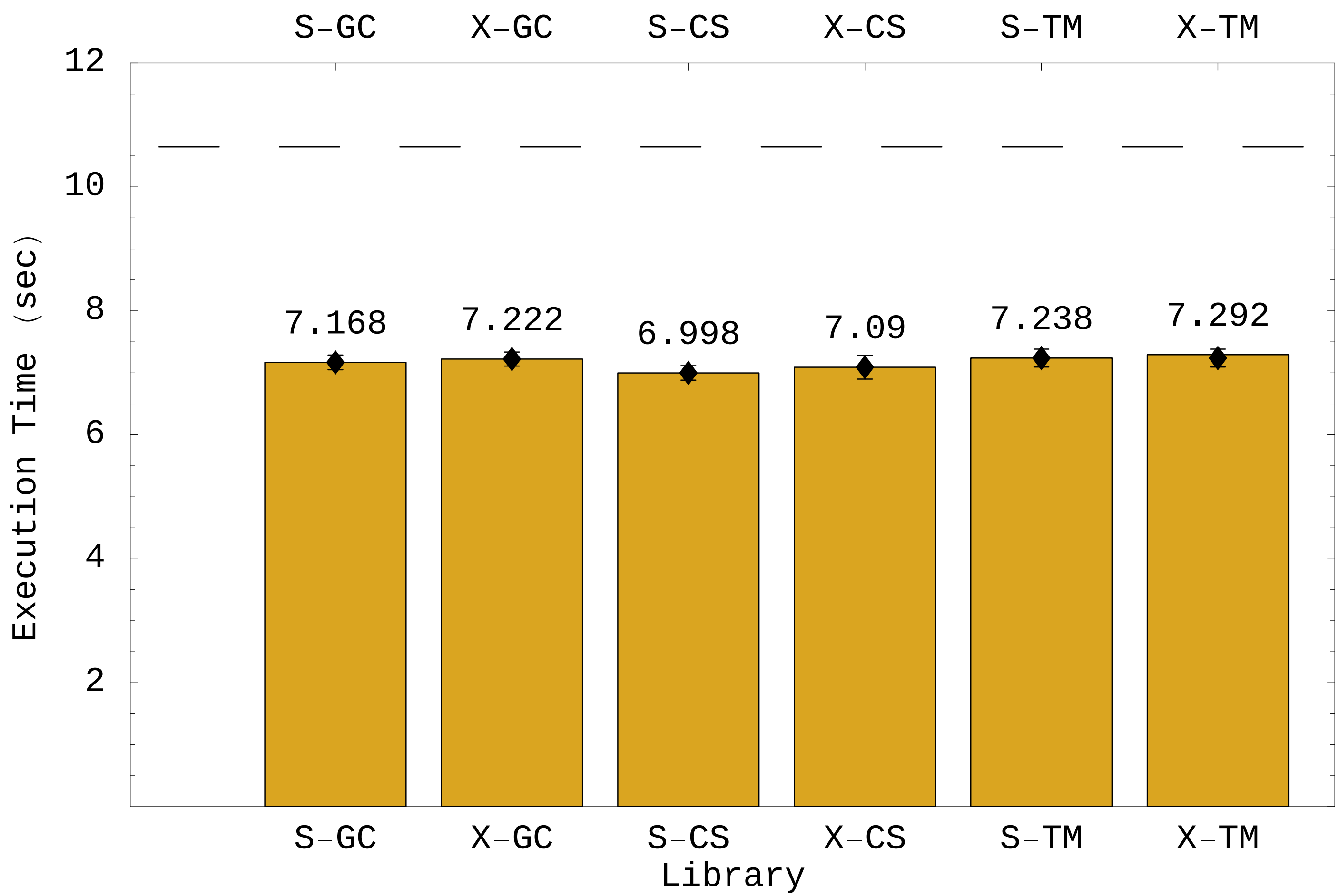
- Testing time increases signifi cantly when memory is *Min*

Summary of Reductions for `Library`

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	32.7	78.8 ✓
X-GC	32.1	65.0
S-TM	32.0	72.8
X-TM	31.5	62.3
S-CS	34.3 ✓	61.4
X-CS	33.4	59.8

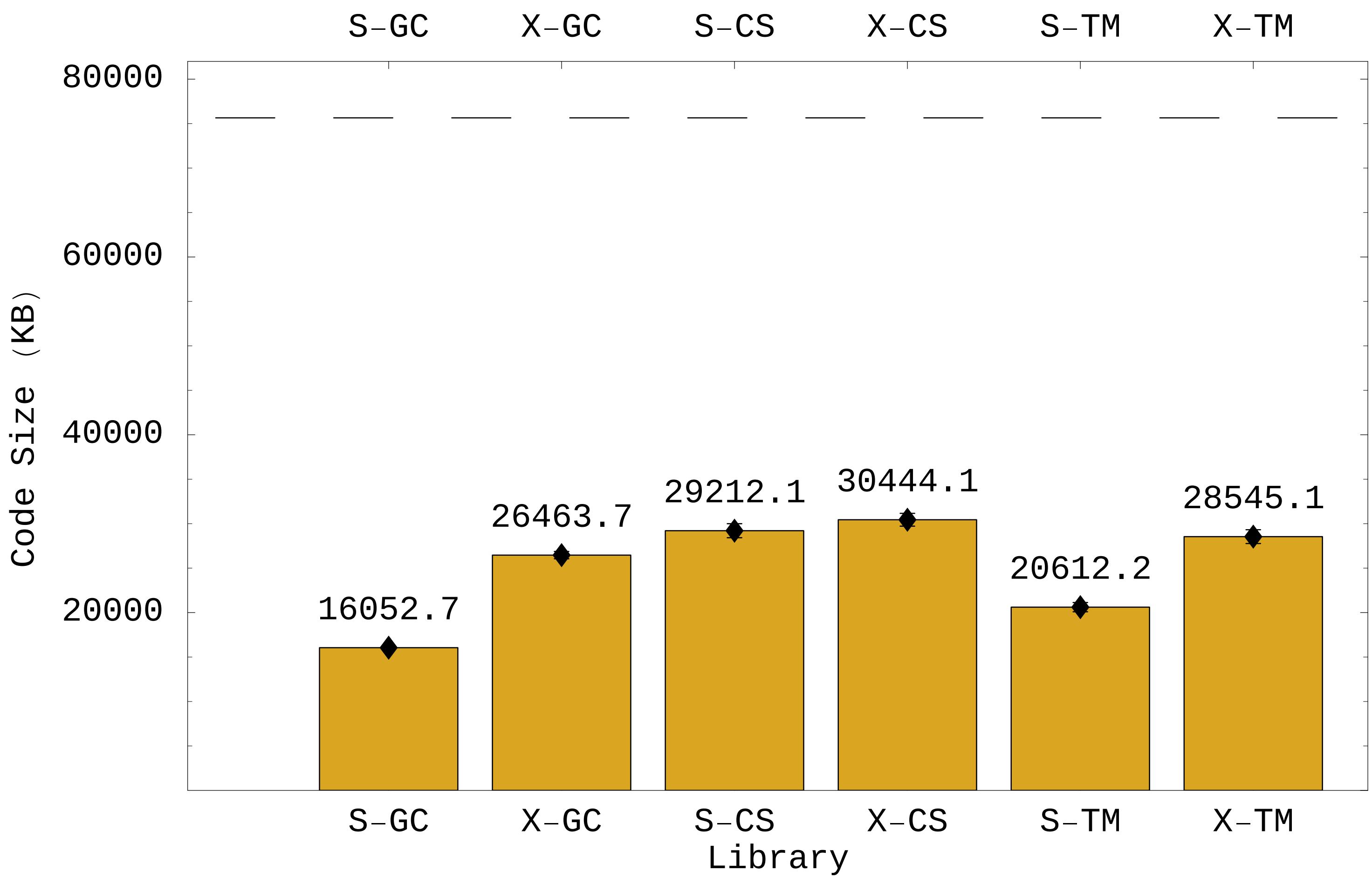
- Signifi cant reductions in time and space required for testing

Testing Time Overhead: Library



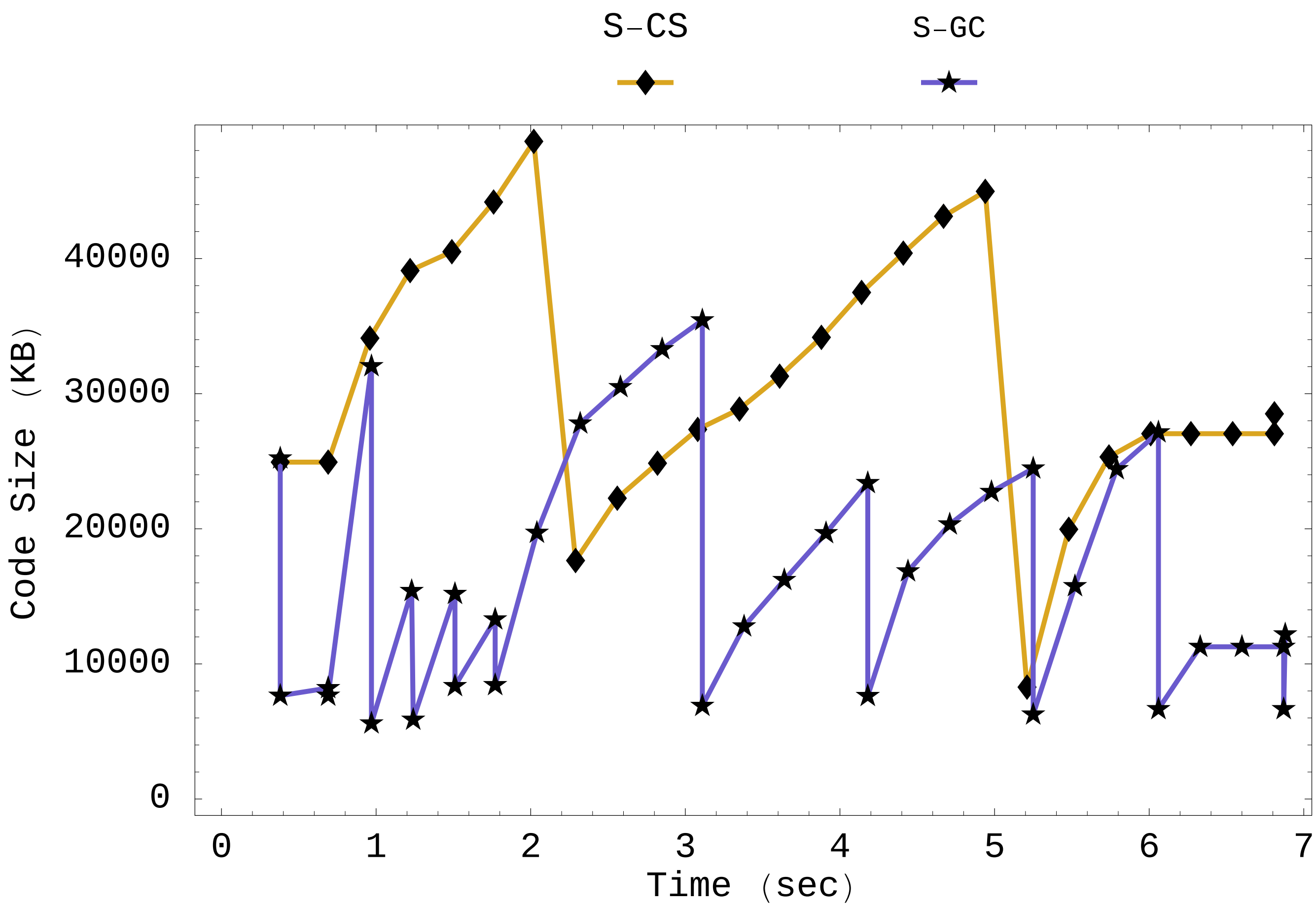
- *S* vs. *X*: Similar reductions for all code unloading techniques

Testing Space Overhead: Library



- Code size reduction does not always yield best testing time

Code Size Fluctuation: `Library`



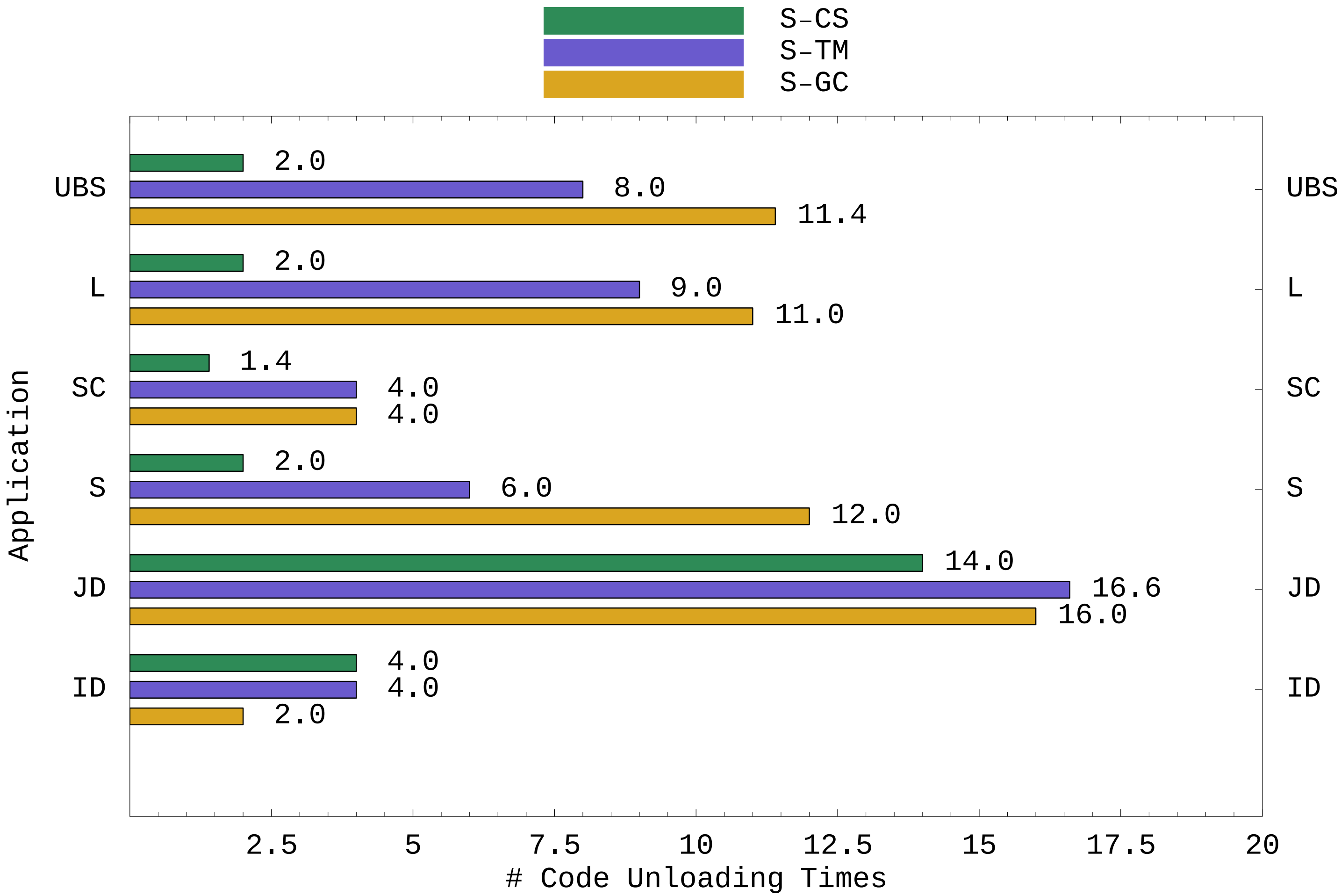
- *S-GC* causes code size fluctuation that increases testing time

Summary of Reductions for `ID`

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	-1.1	42.5
X-GC	-1.1	26.7
S-TM	-1.2	44.5
X-TM	-.29 ✓	28.8
S-CS	-.77	51.4
X-CS	-1.4	61.4 ✓

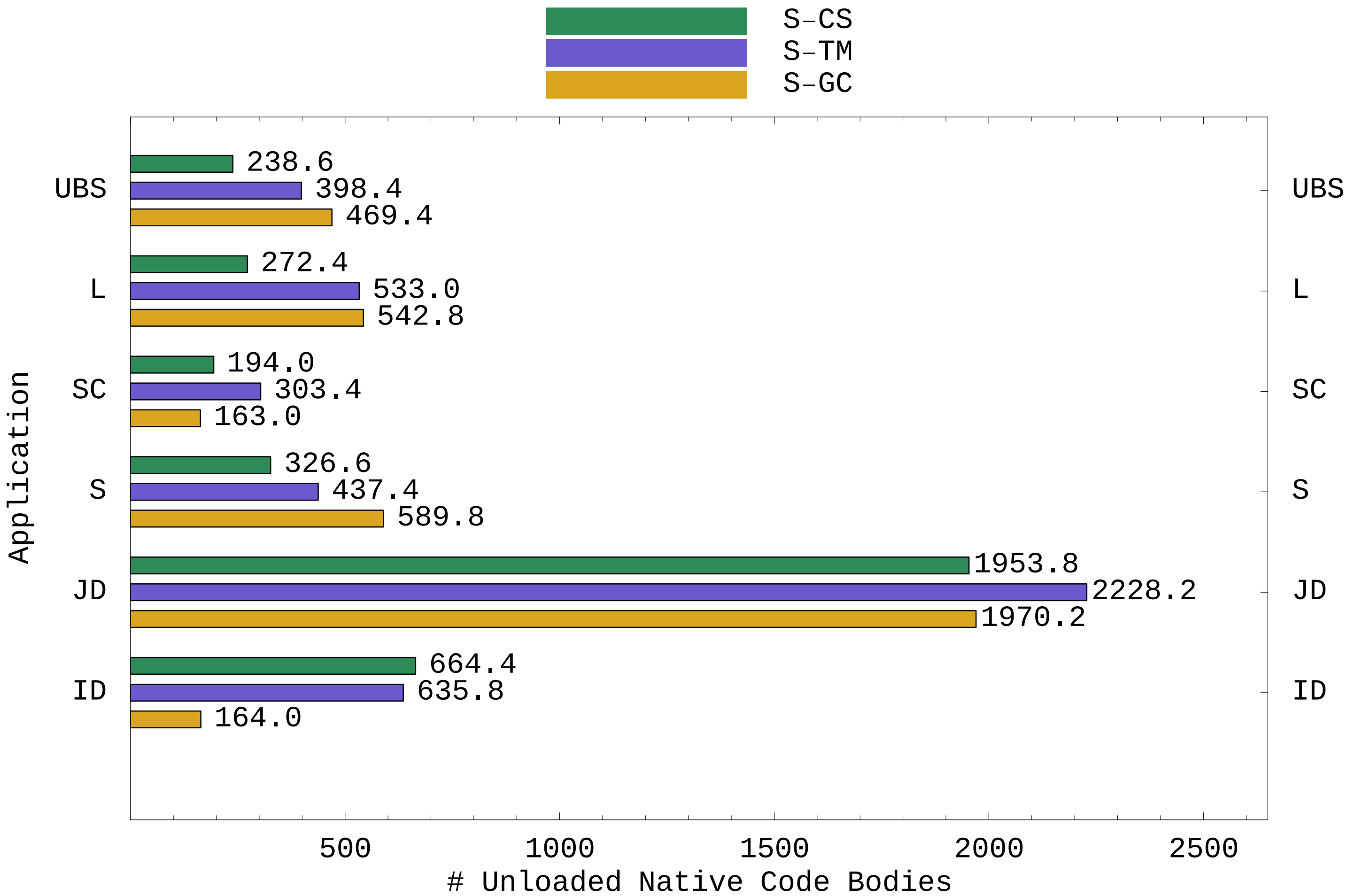
- Unloading can decrease code size while still creating an overall increase in testing time

Number of Code Unloads



- All techniques cause `ID` to perform few unloading sessions

Number of Unloaded Code Bodies



- ID's large working set forces unloading of many code bodies

Summary of Reductions

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	16.1	68.4 ✓
X-GC	16.4	52.8
S-TM	17.1	62.6
X-TM	16.4	45.9
S-CS	17.6 ✓	58.8
X-CS	15.3	54.8

- Across all applications, adaptive code unloading techniques reduce both testing time and space overhead

Conclusions and Future Work

- Dynamic compilation in JVMs can increase testing time if memory is constrained
- Adaptive unloading of native code often reduces memory overhead, avoids GC invocation, and reduces testing time
- Impact of unloading varies with respect to size of application’s working set and program testing behavior
- Regression test suite prioritization and reduction techniques that consider structural coverage *and* time and space overheads

Additional Resources

- Kapfhammer et al. Testing in Resource Constrained Execution Environments. In *IEEE/ACM Automated Software Engineering*. November 7 - 11, 2005.

http://cs.allegheny.edu/~gkapfham/research/juggernaut/